A report on "A Generator for Type Checkers" by H. Gast

Andrey Breslav

November 26, 2009

Abstract

This report presents TCG, a system for generating type checkers from declarative specifications (developed by H. Gast in his PhD thesis in 2004). The system models type checking process as a proof search for typing judgements. It is expressive enough to describe type systems of different kinds of programming languages: from simply-typed λ-calculus to object-oriented imperative languages. The usage of the system is illustrated with a type checker specification for the MINIML language, which captures the key features of ML, including type inference, polymorphic definitions and mutual recursion.

1 Introduction

Tools for generating parsers from different types of context-free grammars have been available for decades. They serve for automating the routine task of developing a parser and make it less error-prone by working with high-level specifications, which are (in most cases, partially) declarative and checkable for consistence.

However, a context-free syntax does not capture all the features of a programming language. There remains the "non-context-free syntax" or "static semantics" that is checked by hand-written code. Since the type theory does not seem underdeveloped compared to automata theory, we should expect availability of type checker generators as well as parser generators. And these tools might be a big advantage for implementing non-context-free static checks.

This report is based on a Ph.D. thesis by H. Gast "A Generator for Type Checkers" [G04]. Chapter 5 of the thesis examines the related work in detail, and we will not go into it here. The only thing we have to mention is that

there were attempts to create such tools before, but most of them suffer from being more of domain-specific *programming* languages than *declarative* specification languages. The TYPE CHECK GENERATOR (TCG) tool described in the thesis makes a significant advance towards the declarative style.

2 Approach Overview

We use types in programming languages to enforce certain properties of our programs at compile time. Types themselves (provided as annotations or inferred) can be thought of as statements about those properties, which the compiler (to be precise, the type checker subsystem of the compiler) ensures in some way.

In TCG the type checking process is modeled as a search for explicit proofs of statements associated with an Abstract Syntax Tree (AST) of a program. The proofs are built with respect to inference rules specified by a user; in fact, the rules form the core part of a declarative specification of a type system in TCG. This approach is, of course, related to the Curry-Howard correspondence, but only in a broad sense: TCG builds proofs using the AST structure but not necessarily strictly according to it.

A language of types used in TCG is limited only by very general requirements like having most general unifiers (for formal definitions see Chapter 2 of [G04]). Strictly speaking, we cannot always identify a type within a TCG statement, since a *typing relation* $(e : t)$ does not have to be present in it. The types, where present, may have almost arbitrarily complicated structure. This structure, as well as the structure of proofs, is described by *typing rules* (specified by a user) which are basically inference rules of a proof system.

A generic typing rule consists of a set of universally quantified *variables*, a list of *premises*, a *conclusion* and a list of *parameters*:

$$\forall\,(variables)\ \frac{premise_1 \quad \ldots \quad premise_n}{conclusion}\ (\texttt{rule\_name}[\textbf{parameters}])$$

Beside the typing rules themselves, a TCG specification describes

- context-free syntax;
- abstract syntax tree (AST) construction;
- pretty-printing rules.

The latter is irrelevant for our discussion.

At generation time (Figure 1) the translator produces a parser and an initial context for the proof search.

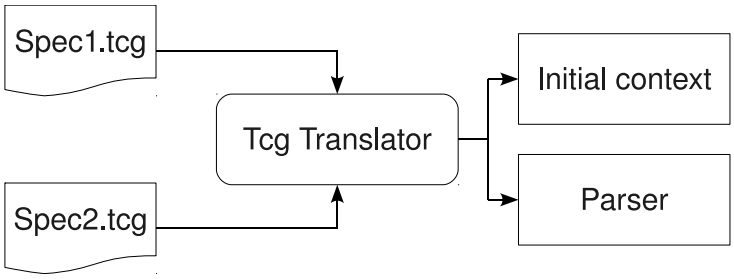

Figure 1: The Tcg Generator

At runtime (Figure 2) the parser builds an AST and collects associated goals to be proven. Then it runs an "interpreter" which searches for proofs of the collected goals.

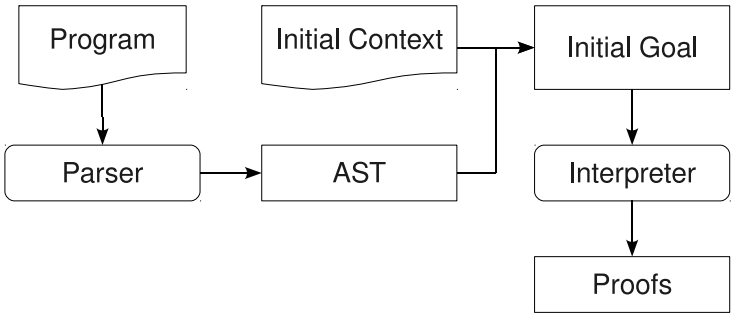

Figure 2: The Tcg Type Checker at Run Time

The basic proof search works roughly as follows: it starts with an *initial goal* (see Figure 2), a term which has to be proven; then, it looks for a typing rule which conclusion unifies with the goal; if such a rule exists, it is *instantiated*: the quantified variables are bound and the rule application marks a node in a proof tree, the premises of the rule become new goals to be proven. If for some goal there's no rule applicable, the procedure backtracks.

This process basically implements a backwards reasoning with respect to the inference rules. To make the procedure more efficient and powerful, Tcg provides a number of extensions to it, such as passing parameters to rules, which the user may trigger by referring to them explicitly in the rule definitions (these techniques are illustrated below). The formal proof calculus which provides a theoretical foundation for the proof search is described in Chapter 2 of [G04].

In this report we mainly illustrate the usage of Tcg by giving examples of typing rules (aiming at implementing a type checker for the MiniML language [CDDK86] as it is done in Chapter 4 of [G04]). Since Tcg rules

include proof search instructions, the same set of examples illustrates the main techniques used to direct the proof construction. Formal definitions of the rule language are given in Chapter 3 of [G04].

3 Tcg In Action

Context-free syntax in Tcg is described by YACC-style productions; each grammar must have a start symbol called `file`. Productions may be annotated with AST building instructions (after `-->` sign) or a block of code to be executed after the production has been matched (in {! ... !} brackets). The `file` symbol is responsible for running type analysis, which is usually done by invoking the `run` command:

```
file: top_wrap EOF {! run ~save:
                        [ (input.base^".rls",
                            [ ("define", "save_defined") ]) ] $1
                    !}
```

The arguments passed to the `run` command deal with saving results and are irrelevant for our discussion.

The rest of this section presents the case study given in Chapter 4 of the original work [G04].

3.1 Simply-Typed λ-Calculus

The context-free syntax of the simply-typed λ-calculus is described by the following rules (we omit lexical definitions and documenting annotations):

```
top_wrap: tops          --> tops($1)

tops: top               --> $1 :: []
    | top ";;" tops     --> $1 :: $3

top: exp                --> $1

exp: ID                 --> id[$1]
   | exp exp            --> apply($1,$2)
   | "\\" ID "." exp    --> lambda(id[$2],$4)
   | "(" exp ")"        --> $2
```

The first three rules describe a sentence as a list of expressions (`exp`) separated by double-semicolons ("`;;`"). The last rule describes a λ-expression in a usual way: as a variable, application, abstraction or bracketed expression. As mentioned before, instructions after the "`-->`" symbols serve for AST-construction; there are several types of such instructions:

- item references of the form `$<number>` — reference items on the left-hand side;
- node constructors of the form `<node_type>(<children>)`;
- list constructors of the form `head::tail`, where `[]` denotes the empty list;
- opaque term constructors of the form `<class>[<value>]` – these will be explained later.

For example, a textual representation of the expression

$$(\lambda f.\lambda x.f\ x)\ g\ y$$

will be

```
(\f.\x.f x) g x,
```

and an AST constructed from it is

```
tops(
    apply(
        apply(
            lambda(id[f],
                lambda(id[x],
                    apply(id[f], id[x]))),
            g),
        x)
    ::[]
)
```

We will refer to pairs of the form

$$term : type$$

as *typing judgements*, where *term* is an AST subtree and *type* is a term structured according to typing rules. The typing rules are used by Tcg to construct proofs of typing judgements, which are basically trees where the

proven judgement is a root and all the edges are *properly* labeled with rules (what "properly" means will be explained later).

As we said before, a generic typing rule consists of a set of universally quantified *variables*, a list of *premises*, a *conclusion* and a list of *parameters*. In TCG's concrete syntax it is written like this:

```
rule apply
  forall(f,e,s,t)
    apply(f,e) : t
  if f : fun(s,t)
  and e : s
```

In this example, `f`, `e`, `s` and `t` are the quantified variables, `apply(f,e)` is the conclusion, `f:fun(s,t)` and `e:s` are the premises. In the more familiar notation of inference rules it can be written like this:

$$\forall\,(f,e,s,t)\ \frac{\Gamma \vdash f : s \to t \qquad \Gamma \vdash e : s}{\Gamma \vdash f\ e : t}\ (\texttt{apply})$$

This rule expresses the typing for an application. To construct a proof for a judgement $f\ e : t$ we have to build a tree, where this judgement will be the root node and its children will be obtained by *application* of this rule. So the root node will have two children having the forms of the premises, where f, e and t are already known (they appear in the initial judgement) and s must be inferred. Each node in the tree is marked with the rule applied at this node, in our case the root node will be marked with `apply` rule. To complete the proof we must construct subtrees proving the children, so that leaves of the tree will be marked with application of rules which have no premises.

We have written Γ in the example above to provide the most familiar notation, but in TCG the context is "global" for the proof search and is being modified each time a rule is applied (the modified version is relevant to the subtree of the vertex marked with the applied rule). This is motivated by the backward style of reasoning adopted by TCG: when we apply a rule, we create new vertices in the proof tree and change our knowledge about currently available typing information. These changes are expressed by *context modifiers* which are written in the premises instead of context variables like Γ. For example, here is the rule for λ-abstraction:

```
rule lambda
  forall(x,e,s,t)
    lambda(x,e) : fun(s,t)
  if e : t
    under -( :.1.= x) + [ x : s ]
```

Here is the tree-like form of this rule:

$$\forall\,(x,e,s,t)\ \ \frac{-(:_1{=}\,x),+[x:s]\vdash e:t}{\lambda x.e:fun(s,t)}\ (\texttt{lambda})$$

The context modifiers here are `-(:.1. = x)` and `+[ x : s ]`. The first one removes rules assigning a type to x from the context of the conclusion, it consists of two parts: a minus sign, which denotes removal and a *selector*

$$:_1{=}\,x$$

which is basically a pattern meaning that the top function symbol in terms we are going to delete is a colon and its first argument is x. Another context modifier in this example adds a new typing fact: "$x : s$".

The context modifiers are applied to the context available at the root of the subtree, so they are relative to the context of the conclusion, that is why we do not write anything denoting context under the like.

The rule `apply` mentioned above has no context modifiers, thus in tree-like form it is written as follows:

$$\forall\,(f,e,s,t)\ \ \frac{\begin{array}{c}\vdash f:s\to t\\ \vdash e:s\end{array}}{fe:t}\ (\texttt{apply})$$

The third rule for simply-typed λ-calculus will be the following:

$$\forall\,(es,ts)\ \ \frac{\vdash[\texttt{branch}]\ es:ts\ \mathbf{export}*\ldots}{tops(es)}\ (\texttt{tops})$$

Note that the conclusion here is not a typing judgement; there is no problem: we are basically proving a theorem and our language is not limited to a single predicate ":". The premise contains a `[branch]` modifier which denotes that it must be looked for by a independent sub-search procedure; the ellipsis ... is a shorthand denoting list iteration: in our AST structure `tops` contains a list of children, so the `es` variable will iterate through it item-by-item. The meaning of **export** directive will be explained later.

The final step is to construct an initial set of rules, which are available at the beginning of the proof search. We put all our rules there:

```
environment apply,lambda,tops
```

3.2 Introducing Constants

Assume we want to have integer or boolean literals in our language. These will by represented by corresponding tokens: `INT`, "`true`" and "`false`". AST productions for these will be the following:

```
  INT     --> int[$1]
| "false" --> bool["false"]
| "true"  --> bool["true"]
```

As we have seen before, here *opaque* terms, e.g. `int[$1]`, are constructed instead of normal AST nodes, these terms are opaque for the type checker: it can not look inside them, and can only consider their classes (`int` and `bool` in our case), but the values can be extracted for output (e.g., for error reporting).

Typing rules for such constants will be of the following form:

$$\forall\,(i)\ \frac{}{\texttt{int}[i] : \texttt{int}}\ (\texttt{int\_const})$$

We can declare "primitive" or "built-in" operations in the same manner:

$$\frac{}{add : int \to int \to int}\ (\texttt{add\_function})$$

And likewise are conditional expressions:

$$\forall\,(e_1, e_2, e_3, t)\ \frac{\begin{array}{c}\vdash e_1 : bool \\ \vdash e_2 : t \\ \vdash e_3 : t\end{array}}{\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ endif} : t}\ (\texttt{if\_expr})$$

3.3 Bindings

For the monomorphic version of let expression (`let x = e in e'`) it is sufficient to treat it as λ abstraction applied to the bound term: $(\lambda x.e')\ e$. Thus we can write down the following rule:

$$\forall\,(x, e, e', s, t)\ \frac{\begin{array}{c}\vdash e' : s \\ -(:_1= x), +[x : s] \vdash e : t\end{array}}{\textbf{let } x = e' \textbf{ in } e : t}\ (\texttt{mono\_let})$$

In case of polymorphism, which means that x may be typed with $\forall\widetilde{\alpha}.s$, this approach will not work directly, since we need to quantify all the *inner variables* of the type scheme s, namely the variables which appear freely in

the proof of $e' : s$ and do not appear in other parts of the proof tree. The operators presented by now do not allow to do this, so TCG introduces tools which are expressive enough to cover it (and much more): *rule extraction* and *forward application*. To illustrate these techniques, we will first reformulate the above rule `mono_let` and then extend it to a polymorphic version.

Rule extraction basically takes a (possibly incomplete) subproof and turns it into a new rule: the root of the subproof becomes a conclusion, and the leafs become premises. If the subproof was complete, then there will be no premises in the rule (all the premises will be empty). This operation corresponds to the idea of memorizing subproofs as lemmata and re-using them in different parts of the proof. Thus we are going to extract a proof of $e' : s$ and then use it in the proof for the second premise (and later for the proof of $\forall x : \widetilde{\alpha}.s$).

The newly extracted rule helps in proving $e' : s$, but what we need for the second premise is $x : s$. This is where *forward application* comes into the stage: it allows to combine two rules into one if the premise of the second rule unify with the conclusion of the first one.

$$\left.\begin{array}{c} \dfrac{\cdots}{e' : s} \\ + \\ \forall\,(e,t)\ \dfrac{e : t}{x : t} \end{array}\right\} \Rightarrow\ \frac{\cdots}{x : t}$$

This operation easily generalizes to more than one premise of the second rule.

Now we can reformulate `mono_let` in this manner: in the context modifier for the second premise we will extract the proof of the first premise and perform the forward application, which joins the extracted subproof with the auxiliary rule `let_binding`.

$$\forall\,(x, e, e', s, t)\ \frac{\begin{array}{l}\vdash e' : s \\ -(:_1 = x), +[\texttt{let\_binding}[x]](\langle 1 \rangle) \vdash e : t\end{array}}{\textbf{let}\ x = e'\ \textbf{in}\ e : t}\ (\texttt{let\_subproof})$$

The auxiliary rule `let_binding` looks like this:

$$\forall\,(e, t)\ \frac{\vdash e : t}{\mathbf{y} : t}\ (\texttt{let\_binding}[\mathbf{y}])$$

This rule is not used on its own, only with forward application; thus, it is not a normal inference rule for the proof system.

Let us explain the new pieces of notation now. First, we have a *rule reference* in the context modifier of the first premise:

$$\texttt{let\_binding}[x]$$

this simply adds the rule `let_binding` to the context, passing x to it as an argument. The rule has a formal parameter $\mathbf{y}$; this parameter must be instantiated with a term upon the rule application; in our case it is instantiated with the term x.

The rule reference is followed by a forward application:

$$[\texttt{let\_binding}[x]](\langle 1 \rangle)$$

this notation means "extract the proof of the first premise ($\langle 1 \rangle$) and perform forward application on it and the rule `let_binding`$[x]$". The generic syntax for forward application operation will be

$$\mathit{rule\_expression}(\mathit{rule\_expressions})$$

where each rule expression may be

- an inline rule (e.g., $x : s$),
- a reference (e.g., `let_binding`$[x]$),
- a rule extracted from the subproof of the premise number i (e.g., $\langle 1 \rangle$, the premise must stay to the left of the current premise),
- the **environment** which refers to all the rules present in the context.

Now we proceed to the polymorphic version of **let**. To complete the definition, we change `let_subproof` only slightly:

$$\forall\,(x, e, e', s, t)\ \frac{\begin{array}{l}\vdash e' : s \\ -(:_1= x), +[\texttt{let\_binding}[x]](\langle 1 : [\forall] \rangle) \vdash e : t\end{array}}{\textbf{let}\ x = e'\ \textbf{in}\ e : t}\ (\texttt{let\_poly})$$

Here $\langle 1 : [\forall] \rangle$ stands for "extract the subproof of the first premise and quantify its inner variables universally". These will be exactly $\widetilde{\alpha}$ — the variables quantified in the type scheme $\forall\widetilde{\alpha}.s$. Note that the type scheme itself does not appear in the rule: we do not need to enrich our predicate language with universal quantification since it is successfully handled by the meta-theory. Why is the meta-theory designed in such a way that it serves exactly this purpose? The author of the original work gives a detailed explanation in the Section 4.1.3.3, to put it shortly: quantifying the free variables is a natural operation which makes a rule usable, and inner variables are the maximal set of variables which can be quantified without affecting soundness.

3.4 Recursion

To define non-trivial functions with **let**, we need recursion, which is not yet supported by our rule. To support it we can just add a context modifier to the first premise:

$$\forall\,(x, e, e', s, t)\ \frac{\begin{array}{l}+[x : s] \vdash e' : s \\ -(:_1= x), +[\texttt{let\_binding}[x]](\langle 1 : [\forall]\rangle) \vdash e : t\end{array}}{\textbf{let}\ x = e'\ \textbf{in}\ e : t}\ (\texttt{let\_poly})$$

By doing this we have added the assumption $x : s$ to the proof for $e' : s$, and since x is being bound to e' this indeed enables recursion.

By now we can handle all the basic functionality available in a functional language. In further subsections we illustrate more sophisticated features of Tcg by extending the capabilities of the language with more convenient mechanisms like multi-binding `let` and mutual recursion.

3.5 Parallel bindings

To make our language more practical, let us introduce `let` expressions binding many variables at the same time. The syntax will be the following:

```
exp: "let" bind_group "in" exp    --> let($2,$4)

bind_group: bind                  --> $1 :: []
          | bind "and" bind_group --> $1 :: $3

bind: ID "=" exp                  --> bind(id[$1],$3)
```

In the AST a bind group is represented by a *list* of bindings. Now we cannot write a rule of the from

$$\frac{\begin{array}{l}\vdash \textit{typings of bindings} \\ +[\texttt{let\_binding}[x]](\langle \textit{bindings} : [\forall]\rangle) \vdash e : t\end{array}}{\textbf{let}\ \textit{bindings}\ \textbf{in}\ e : t}$$

because *typings of bindings* is not a single premise, but a list of variable length. Instead, we can process the list by usual recursion:

$$\frac{\vdash \textit{head} \quad \vdash \texttt{bind\_group}(\textit{tail})}{\texttt{bind\_group}(\textit{head} :: \textit{tail})} \qquad \frac{}{\texttt{bind\_group}([])}$$

Then, the `let` rule will look like this (by ??? we deonte parts of the rule which will be refined later):

$$\forall\,(bs, e, t)\ \frac{\begin{array}{c}\vdash \texttt{bind\_group}(bs) \\ +[???](\langle ??? : [\forall]\rangle) \vdash e : t\end{array}}{\textbf{let}\ bs\ \textbf{in}\ e : t}$$

By now we did not show what to extract from the premise and what to compose it with (by forward application). Indeed, what we did in the previous example was extracting a subproof which ended with a typing judgement $e' : s$, but here the premise is not a typing judgement, judgements are "hidden" in its proof:

$$\frac{\begin{array}{c}e'_1 : s_1\ \ldots\ e'_n : s_n \\ \vdots \\ \vdash \texttt{bind\_group}(bs) \qquad \ldots(\langle proofs\ for\ e_1 \ldots e_n : [\forall]\rangle) \vdash e : t\end{array}}{\textbf{let}\ bs\ \textbf{in}\ e : t}$$

TCG provides a generalized for of rule extraction which is useful here: the rules may be marked with "**export** *label*", the label will be put on the proof tree nodes; the extracting operator may be directed to extract not the whole tree but only the subtrees marked with a certain label. Not all the label in a proof tree are visible from its root, but only those which were *propagated* up the tree. If a node in the tree is marked with a special label "$*$", it propagates labels of all the child nodes up. To be able to extract all the needed typing judgements from the proof of $\texttt{bind\_group}(bs)$, we can do the following:

$$\frac{\vdash head\ \textbf{export}\ \texttt{bind} \qquad \vdash bind\_group(tail)\ \textbf{export}\ *}{bind\_group(head\ \texttt{::}\ tail)} \qquad \frac{}{bind\_group(\texttt{[]})}$$

By doing this, we label every typing judgement from bs and propagate these labels up the tree when doing a recursive call. This situation is rather common, so TCG provides a shorthand for it: to handle lists we can use ellipsis $(\ldots)$ at the premise, this will tell the system to generate the two rules given above. Now we can complete the initial rule:

$$\forall\,(bs, ts, e, t)\ \frac{\begin{array}{c}\vdash bs : ts\ \textbf{export}\ \texttt{bind}\ \ldots \\ +[\texttt{exp\_binding}](\langle 1 : \texttt{bind}[\forall]\rangle) \vdash e : t\end{array}}{\textbf{let}\ bs\ \textbf{in}\ e : t}$$

The extraction operator is now $\langle 1 : \texttt{bind}[\forall]\rangle$, which will extract from the proof of the first premise all the subproofs labeled with `bind` and universally quantify the inner variables *of the whole proof* of the first premise (this

"over-quantification" is safe). Note that the variable ts is free, so it will be quantified. The auxiliary rule `exp_binding` is the following:

$$\forall\,(x, e, t)\ \frac{\vdash x = e : t}{x : t}\ (\texttt{exp\_binding})$$

We also need a rule to be able to rove judgements of the form $x = e : t$. It is straightforward:

$$\forall\,(x, e, t)\ \frac{\vdash e : t}{x = e : t}\ (\texttt{binding})$$

3.6 Mutual Recursion

To type-check a mutually recursive definition like this

```
letrec even = \x. if (eq_int x) 0 then true else odd ((sub x) 1)
   and odd = \x. if (eq_int x) 0 then false else even ((sub x) 1)
in even 5
```

we need to choose fresh type variables α_1 and α_2 and assume $even : \alpha_1$ and $odd : \alpha_2$ before the right-hand sides of the bindings. This procedure is different from what we have seen by now in a way that it needs "a lookahead", in other words, we cannot look at every item in the program only once. To handle this, we write the following rules:

$$\forall\,(bs, e, t, x)\ \frac{\begin{array}{l}\vdash \mathit{bind\_group}(bs)\ \mathbf{export}*\\ -(:_1= x \rightarrow :_1= x); +[\texttt{exp\_binding}](\langle 1 : \texttt{bind}[\forall]\rangle) \vdash\\ e : t\end{array}}{\mathbf{letrec}\ bs\ \mathbf{in}\ e : t}\ (\texttt{letrec})$$

$$\forall\,(bs, ts, x)\ \frac{\begin{array}{l}\vdash bs : ts\ \mathbf{export}\ \texttt{fwd}\ \ldots\\ -(:_1= x \rightarrow :_1= x); +[\texttt{exp\_binding}](\langle 1 : \texttt{fwd}\rangle) \vdash\\ bs : ts\ \mathbf{export}\ \texttt{bind}\ \ldots\end{array}}{\mathit{bind\_group}(bs)}\ (\texttt{bind\_group})$$

The first premise of the rule `letrec` contains the most significant information: all the recursive definitions are checked there. The check itself is done by the `bind_group` rule, which iterates over the first premise, checks each element in bs and marks the result with `fwd` label. These results will have the form $x = e : \alpha$ for a fresh α since ts does not appear anywhere else in the rule, thus there is nothing to prove about such judgements, and we can make TCG's life easier by marking them as *solved goals* (done by putting "!" after the `fwd` label). The overall process precisely mimics what was discussed

for the code example above: we take fresh type variables, assume typings for bound variables and check the expressions to the right of "=".

The context modifier $-(:_1= x \rightarrow :_1= x)$ is a *guarded* one: it is triggered by adding a rule matching $:_1= x$ (to the left of the arrow) is added, namely, it removes all the previously existing bindings for x when a new binding is added.

4 Conclusion

We have illustrated the usage of TCG by developing a type system for the MINIML programming language, which supports mutually recursive polymorphic functions. While doing this we have used most features of the generator, so the reader has had a chance to overview the system. Our examples have shown that the specification language is purely declarative and rather concise.

The original paper [G04] provides many interesting examples, which we did not cover, such as imperative languages, object-oriented languages and a language for stating generic algorithms with explicit requirements on their parameter types. These examples show that the approach works for large variety of type systems, not only functional languages with Hindley-Milner type inference.

The original paper also provides comparison with other approaches to type system specification (Chapter 5), such as TYPOL [BCD89], TINKERTYPE [LP03], constraints solving and logical frameworks (ISABELLE [P94]). These comparisons bring the evidence of TCG being, on the one hand, much less of a (logical) programming languages than previous solutions, on the other hand, an fully specialized integrated framework as opposed to constraint solvers and logical frameworks. On the "technical" level the ability of subproof extraction is a distinguishing feature of TCG.

A major disadvantage which may prevent this approach from being used in practice might be the lack of comprehensible error reporting: when no proof is found (which means that the program is not correct), a type checker must complain with a helpful error message mentioning the reason of the problem; this aspect was not studied in the [G04].

References

[G04] H. Gast, A Generator for Type Checkers, *Ph.D. Thesis*, Eberhard-Karls-Universität Tübingen, 2004.

[CDDK86] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A Simple Applicative Language: Mini-ML. *In Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 1326, Cambridge, Massachusetts, USA, August 1986. ACM.

[BCD89] P. Borras, D. Cle ment, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. *ACM SIGPLAN Notices*, 24(2):1424, February 1989.

[LP03] M. Levin and B. Pierce. TinkerType: A language for playing with formal systems. *Journal of Functional Programming*, 13(2), March 2003.

[P94] L. Paulson. Isabelle A Generic Theorem Prover. *Number 828 in Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg, 1994.